(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
23 May 2002 (23.05.2002)

PCT

(10) International Publication Number
WO 02/41145 A2

(51) International Patent Classification⁷: G06F 9/45

(21) International Application Number: PCT/CA01/01090

(22) International Filing Date: 27 July 2001 (27.07.2001)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
60/252,170 20 November 2000 (20.11.2000) US
09/746,016 26 December 2000 (26.12.2000) US

(71) Applicant (*for all designated States except US*): ZUCOTTO WIRELESS INC. [US/US]; Suite 400, 4225 Executive Square, La Jolla, CA 92037 (US).

(72) Inventor; and
(75) Inventor/Applicant (*for US only*): BOTTOMLEY, Thomas, Mark, Walter [CA/CA]; 803 Foxwood Court, Orleans, Ontario K4A 3E9 (CA).

(74) Agents: BRETT, R., Allan et al.; Smart & Biggar, 900-55 Metcalfe Street, P.O. Box 2999, Station D, Ottawa, Ontario K1P 5Y6 (CA).

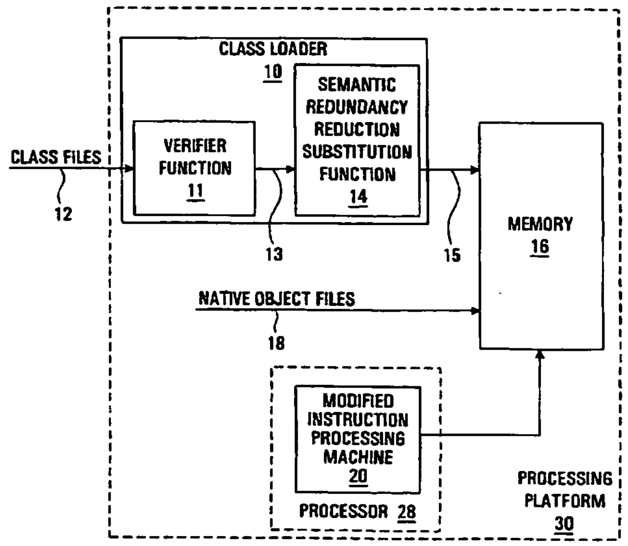
(81) Designated States (*national*): AE, AG, AL, AM, AT, AU, AZ, BA, BB, BG, BR, BY, BZ, CA, CH, CN, CO, CR, CU, CZ, DE, DK, DM, DZ, EC, EE, ES, FI, GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MA, MD, MG, MK, MN, MW, MX, MZ, NO, NZ, PL, PT, RO, RU, SD, SE, SG, SI, SK, SL, TJ, TM, TR, TT, TZ, UA, UG, US, UZ, VN, YU, ZA, ZW.

(84) Designated States (*regional*): ARIPO patent (GH, GM, KE, LS, MW, MZ, SD, SL, SZ, TZ, UG, ZW), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE, TR), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, ML, MR, NE, SN, TD, TG).

Published:
— *without international search report and to be republished upon receipt of that report*

[Continued on next page]

(54) Title: SYSTEM AND METHODS PROVIDING RUNTIME BYTE CODE SIMPLIFICATION FOR PLATFORM INDEPENDENT LANGUAGES



(57) Abstract: Methods and devices are provided which transform a sequence of instructions of a platform independent instruction set, such as the Java instruction set for example, by defining a group of functionally equivalent instructions in the instruction set and defining a substitution instruction, and by substituting the substitution instruction for each occurrence in the sequence of instructions of one of the predetermined group of functionally equivalent instructions. The methods are extendable to perform substitutions for multiple groups of functionally equivalent byte codes. Also provided is a method of replacing multi-byte instructions with single-byte instructions.

WO 02/41145 A2

SYSTEM AND METHODS PROVIDING RUNTIME BYTE CODE SIMPLIFICATION FOR PLATFORM INDEPENDENT LANGUAGES

Field of the Invention

The invention relates to systems and methods providing runtime byte code simplification of platform independent languages, such as Java-like languages.

Description of the Background Information

Typical software systems include an application program run by an operating system on a processor connected to a number of hardware peripherals. In some systems, such as those where Java™ or Java-like languages are employed, there is additionally a virtual machine, for example, a Java virtual machine (JVM) situated between the operating system and the application program. The Java or Java-like code is executed by the virtual machine, essentially a byte-code interpreter. Java™, is a programming language marketed by Sun Microsystems, Inc. Java is an architecture-neutral, object-oriented, multi-threaded language intended for use in distributed environments. It has become tremendously popular among application developers, and its use on handheld and wireless platforms has been growing by leaps and bounds. The Java or Java-like programming language may include, but is not limited to Java 2 Platform, Enterprise Edition (J2EETM), Java 2 Platform, Standard Edition (J2SETM), and Java 2 Platform, Micro Edition (J2METM) programming languages available from Sun Microsystems, all of which are incorporated herein by reference. Both J2SE and J2ME provide a standard set of Java programming features, with J2ME providing a subset of the features of J2SE for programming platforms that have limited memory and power resources (i.e., including but not limited to cell phones, PDAs, etc.), while J2EE is targeted at enterprise class server platforms.

A Java method consists of a sequence of Java byte codes. Such a method is run on a Java machine which when implemented in software is referred to as a Java Virtual Machine or JVM. Each byte code is one byte, and there are 256 possible byte codes. However, instruction sets for some standardized implementations of Java typically use only 204 of these byte codes referred to as "standard byte codes" hereinafter. The instruction set also typically includes over 20 commonly recognized runtime optimization "quick" byte codes. This leaves only about 30 byte codes which are not used in the Java machine execution, and which are available for implementing local processor specific functionality.

The standard byte codes carry semantic information that is used by the verification portion of the Java class loader to check all classes for structure, environment, and content. Byte codes are self contained operations with the exception of those that follow the "wide" byte code. This byte code modifies the operation of the following byte code and the quantity of parameters following in the instruction byte stream. There are defined 12 different byte codes that can be prefixed by wide. These are aload, iload, fload, lload, dload, astore, istore, fstore, lstore, dstore, ret, and iinc.

An implementation of a Java Machine in hardware may require additional instructions to support the many Virtual Machine functions currently implemented on the host processor in a native instruction set and are not possible with the existing Java byte codes. The number of instructions necessary to support the missing functionality is usually more than the available number of unused byte codes, particularly if aggressive optimization using "quick" byte codes is to be employed. Furthermore, the wide byte code and the need for more instructions than will fit into the unused byte code slots require a more complex processor that is more difficult to design and test, and which may draw more power.

Summary of the Invention

Embodiments of the invention provide methods of transforming a sequence of instructions of a platform independent instruction set, such as the Java-like instruction set for example, by defining a group of functionally equivalent instructions in the instruction set and defining a substitution instruction, and by substituting the substitution instruction for each occurrence in the sequence of instructions of one of the predetermined group of functionally equivalent instructions. The methods are extendable to perform substitutions for multiple groups of functionally equivalent instructions. The sequence of instructions in some embodiments is a sequence of byte codes defining a Java-like method.

Advantageously, by replacing the instructions of a group of functionally equivalent instructions with a single substitution instruction, (which in some embodiments is one of the group of functionally equivalent instructions) instructions are freed for other purposes. For the Java-like embodiment for example, instructions are freed for custom local implementations, or to function as "quick" byte codes.

In one embodiment, the substitution is performed in the process of performing class loading.

In some embodiments, after processing each instruction, the method is adapted to skip over any subsequent portion in the sequence of instructions which contains an argument of the instruction, and continuing with processing a subsequent instruction following the portion in the sequence which contains an argument of the instruction.

In another embodiment, the method further involves defining for at least one group of functionally equivalent multi-instruction width instructions (for example a group of multi-byte instructions where nominal instruction width is a single byte) a respective substitution single-instruction-width instruction. Then, for each occurrence in the method of a multi-instruction-width instruction in a respective one of the at least one group of functionally equivalent multi-instruction-width instructions, substituting the respective substitution single-instruction-width instruction defined for the respective one of the at least one group of functionally equivalent multi-instruction-width instructions. Single instruction-width instruction substitution may also be performed for single multi-instruction-width instructions.

Advantageously, this allows the elimination of multi-byte instructions being fed to a single byte instruction processing machine (a Java Machine for Java instructions), and simplifies processor design.

Another broad aspect provides an apparatus adapted to transform a sequence of instructions of a platform independent instruction set. The apparatus has a memory containing a definition of a group of functionally equivalent instructions in the instruction set and definition of a substitution instruction. The apparatus also has a processor adapted to substitute the substitution instruction for each occurrence in the sequence of instructions of one of the predetermined group of functionally equivalent instructions. In the event the substitution is to occur during class loading, the apparatus also has a class loading function executed by the processor, the class loading function having an input for receiving the sequence of instructions and having an output comprising a modified sequence of instructions in which the substitution instruction has been substituted.

In some embodiments, the memory has a mapping from each possible input byte code to a corresponding transformed byte code, the corresponding transformed byte code being the same as the input byte code in the event no substitution is to be performed. The processor is adapted to process each byte code in a Java-like method in sequence by looking up the transformed byte code in the mapping, and replacing the byte code with the transformed byte code. The memory may also contain for each possible input byte code, an associated data entry indicating how many bytes should be skipped in order to arrive at a subsequent byte code. IN this case, the skips over argument data to a subsequent byte code using the data entry in the memory associated with the input byte code. These embodiments may be adapted to process multi-byte instructions as described previously.

Other embodiments provide a microprocessor, a computer readable medium, firmware, software, and suitable any combinations of such devices adapted to implement any of the methods described herein.

Brief Description of the Drawings

Preferred embodiments of the invention will now be described with reference to the attached drawing in which:

Figure 1 is a block diagram of a Java byte code processing environment according to an embodiment of the invention.

Detailed Description of the Preferred Embodiments

Referring now to Figure 1, shown is a block diagram of a Java-like byte code processing environment provided by an embodiment of the invention. The environment has a class loader 10 adapted to perform class loading and verification of class files 12 which may for example be Java class files. The class loader 10 has a verifier function 11 and a semantic redundancy reduction substitution (SRRS) function 14. A verified output 13 of the verifier 11 is passed to the SRRS function 14 the function of which is described in detail below. The SRRS block 14 has an output 15 consisting of a sequence of byte codes of a modified instruction set, and this output is written to memory 16. Native object files 18 are also written to memory 16. These are byte code sequences in the modified instruction set which may also make use of custom locally defined and implemented byte codes. Typically, the class loader is a program stored in memory 16, although it may alternatively be implemented in firmware, or in hardware. A modified instruction processing machine 20 implements the modified instruction set in hardware, software, or a combination of hardware and software. The modified instruction processing machine 20 obtains sequences of byte codes from memory 16 for execution. The class files 12 may be obtained internally, or through an external communications link, for example a Blue Tooth wireless link. The native object files 18 typically just internal files, but may be obtained externally as well, though the external interface would need to know not to pass such files through the class loader 10.

In one embodiment, the processing environment of Figure 1 is implemented within a processing platform generally indicated by 30, in which the modified instruction processing machine 20 is implemented in hardware forming part of a processor 28, and the class loader is a function stored in memory 16. The processing platform 30 might for example be a Blue Tooth enabled wireless device.

In the description which follows, it is assumed for the purpose of readability that the language of the class files 12 is Java, and the instructions are Java byte codes. However, it is to be understood that embodiments of the invention may be adapted for use with other platform independent languages and in particular for use with other Java-like languages. In particular, it is understood that with appropriate modifications and alterations, the scope of the present invention encompasses embodiments that utilize other programming languages similar in functionality to Java programming languages, for example C-Sharp by Microsoft Corp.

The class loader 10 performs class loading of Java class files 12 and with the verifier 11 performing verification of these files, steps well known in Java processing environments. The Java class files may include variable and constant definitions, and methods. The verification may include ensuring that instructions in the methods do not result in jumps outside the program space, that variable type usage is consistent, and that only valid byte codes are present for example. After the verification in the verifier 11, many byte codes are functionally equivalent differentiated by only now redundant semantic information. For example, consider the three byte codes {aload, fload, iload}. The byte code aload loads a 32 bit from a local variable to the stack, the byte code fload loads a 32 bit floating point value from a local variable to the stack, and the byte code iload loads a 32 bit integer value from a local variable to the stack. All three of these byte codes load 32 bits from a variable to the stack, and as such what physically needs to be done to implement these three byte codes is identical with the result that all occurrences of these three byte codes can be replaced by any one of the three byte codes or another byte code with the net result of freeing two byte codes for other uses. The verifier 11 needs to be able to distinguish between these three byte codes so that it can perform its verification functions such as making sure that objects are consistently used throughout the entire Java method. For example, the verifier 11 makes sure that a given 32 bit value cannot be used as an integer and then be used as a floating point value elsewhere.

Here is another example of a functionally equivalent byte code group: aload_0, iload_0, and fload_0 all load the 32 bit value from local variable 0 to the top of the stack and all occurrences of these three byte codes can be replaced by any one of the byte codes or another byte code with the net result of freeing two byte codes for other uses.

In order to free many of the byte codes for use by other instructions, according to the invention, groups of functionally equivalent byte codes are replaced with a single byte code after class verification by verifier function 11. Referring again to Figure 1, this is done by the SRRS function 14. The SRRS function 14 is a new module which is added to the class loader 10 after the verifier function 11. All byte codes in the methods of the Java class are scanned and the functionally equivalent byte code groups are replaced by a single functionally equivalent byte code. Preferably, wide byte code sequences are also replaced by a single functionally equivalent byte code.

The following thirty-three groups have been identified to be potentially functionally equivalent byte code groups. It is noted that depending upon a specific implementation of the Java machine, certain byte codes might have additional functionality included beyond that expected by standard Java, and this would possibly result in byte codes which would normally be considered semantically equivalent, not being semantically equivalent.

group_1 = aload, fload, iload

group_2 = aload_0, fload_0, iload_0

group_3 = aload_1, fload_1, iload_1

group_4 = aload_2, fload_2, iload_2

group_5 = aload_3, fload_3, iload_3

group_6 = astore, fstore, istore

group_7 = astore_0, fstore_0, istore_0

group_8 = astore_1, fstore_1, istore_1

group_9 = astore_2, fstore_2, istore_2

group_10 = astore_3, fstore_3, istore_3

group_11 = dload, lload

group_12 = dload_0, lload_0

```
group_13 = dload_1, lload_1
group_14 = dload_2, lload_2
group_15 = dload_3, lload_3
group_16 = dstore, lstore
group_17 = dstore_0, lstore_0
group_18 = dstore_1, lstore_1
group_19 = dstore_2, lstore_2
group_20 = dstore_3, lstore_3
group_21 = if_acmpeq, if_icmpeq
group_22 = if_acmpne, if_icmpne
group_23 = ifeq, ifnull
group_24 = ifne, ifnonnull
group_25 = aconst_null, fconst_0, iconst_0
group_26 = dconst_0, lconst_0
group_27 = aaload, faload, iaload
group_28 = daload, laload
group_29 = fastore, iastore
group_30 = dastore, lastore
group_31 = castore, sastore
group_32 = areturn, freturn, ireturn
group_33 = dreturn, lreturn
group_34 = pop, l2i
group_35 = dneg, fneg
```

The modified Java instruction set has a single selected byte code for implementing the byte codes of each of the above-identified groups. Preferably, the single byte code used to implement a given group of byte codes is one of the byte codes in the group. The SRRS block 14 processes a verified Java class file. Each executable byte code in the verified Java class file is examined. If the executable byte code is in one of the above identified groups of byte codes, a substitution from the executable byte code to the group's corresponding selected byte code is made.

A substitution table is maintained as follows:

Input Byte code(Hex)	Output Byte code(Hex)	Length (Bytes)
00	00H	1
01	41	1
...	...	...
FF	FF	ERR

In the above table, the input byte code is a standard byte code found in a method of a class being loaded. The output byte code is the byte code substituted in place of the input byte code. In the event no substitution is to occur, the output byte code is the same as the input byte code. The length field accounts for any arguments which the byte code might require so that the SRRS function can skip over arguments. Byte codes which are not valid input byte codes have an error indication, for example in the length field.

The substitutions identified above free 48 additional byte codes for utilization as native support byte codes in a hardware implementation of a Java machine.

In another embodiment of the invention preferably implemented in combination with the above described embodiment, each of the 12 wide + byte code sequences is replaced with a newly defined single byte code to remove the "prefix" code anomaly from the instruction set. This is aided by having freed 48 byte codes with byte code group substitution.

The following is an example of the wide byte code sequence substitution:

wide, ret, <indexbyte1>, <indexbyte2> becomes nop, ret_w, <indexbyte1>, <indexbyte2> where ret_w is a new byte code added to the instruction set.

The substitution is performed by adding the nop byte code to pad the total instruction length to be the same as the original avoiding any adjustments to jump and branch offsets in the code of the class file.

Applying the functionally equivalent byte code groups technique described above to the new wide byte codes results in only using 6 new byte codes as follows:

load_32_w = wide + aload or iload or fload = group_A

load_64_w = wide + lload or dload = group_B

store_32_w = wide + astore or istore or fstore = group_C

store_64_w = wide + lstore or dstore = group_D

ret_w = wide + ret

iinc_w = wide + iinc

While preferably implemented in combination with the previous embodiment wherein groups of semantically equivalent multi-byte instructions are grouped together and replaced with a respective single substitution byte code, another embodiment provides for the substitution of a single byte code for a wide byte code performed for any single multiple-byte instruction such as identified above.

With this implementation, the net number of additional byte codes required by the new wide combinations is 5 as the wide byte code is now unused for a resulting saving of 43 byte codes while providing a simpler hardware implementation and more room for new byte codes.

In a modified embodiment, a subset of the byte codes identified above to have semantic equivalents are set aside to have additional functionality, as discussed previously. For this example, astore, astore_0, astore_1, astore_2, astore_3, and areturn are engineered to have their regular Java defined function, and also to perform some garbage collection functionality which makes them no longer semantically equivalent. Preferably, in this embodiment, a further wide byte code is defined as follows: astore_32_w = wide + astore.

In this embodiment, the groups group_6, group_7, group_8, group_9, group_10, group_16, group_32 identified above would have two members instead of three. More generally, it is to be understood that the substitution of a single byte code for any two or more semantically equivalent byte codes is within the scope of one embodiment of the invention.

The modified instruction processing machine implements the modified instruction set, and may include custom implementations of the byte codes which are freed up using one or more of the above described substitution methods.

The substitution of the byte code groups and the wide sequences is implemented after the verification phase of the class loader for our example. It may also be integrated into the verifier stage directly. This invention may also be implemented in a software virtual machine to simplify the internal interpreter loop by adding the same byte code substitution post verification.

<u>Example Implementation of the Present Invention</u>

The following is an example implementation of the invention.

```
/*=============================================================
 * SYSTEM: KVM
 * SUBSYSTEM: Internal runtime structures
* FILE:       zsifilter.c
* OVERVIEW: byte code filter
* Copyright © 2000 Zucotto Systems
*==============================================================*/
/*===============================================================
* Include files
*==============================================================*/
#include <global.h>
#include <stddef.h>
/*===============================================================
* Global variables and definitions.
*==============================================================*/
/* Define substitution codes for Xpresso */
#define ZSI_IF_EQUAL_32                 IF_ICMPEQ
#define ZSI_IF_NOT_EQUAL_32             IF_ICMPNE
#define ZSI_IF_ZERO_32                  IFEQ
#define ZSI_IF_NOT_ZERO_32              IFNE
#define ZSI_ZERO_32                     ICONST_0
#define ZSI_ZERO_64                     LCONST_0
#define ZSI_LOAD_32                     ILOAD
#define ZSI_LOAD_32_0                   ILOAD_0
#define ZSI_LOAD_32_1                   ILOAD_1
#define ZSI_LOAD_32_2                   ILOAD_2
#define ZSI_LOAD_32_3                   ILOAD_3
#define ZSI_LOAD_64                     LLOAD
#define ZSI_LOAD_64_0                   LLOAD_0
#define ZSI_LOAD_64_1                   LLOAD_1
#define ZSI_LOAD_64_2                   LLOAD_2
```

```
#define ZSI_LOAD_64_3              LLOAD_3
#define ZSI_ARRAY_LOAD_32          IALOAD
#define ZSI_ARRAY_LOAD_64          LALOAD
#define ZSI_STORE_32               ISTORE
#define ZSI_STORE_32_0             ISTORE_0
#define ZSI_STORE_32_1             ISTORE_1
#define ZSI_STORE_32_2             ISTORE_2
#define ZSI_STORE_32_3             ISTORE_3
#define ZSI_STORE_64               LSTORE
#define ZSI_STORE_64_0             LSTORE_0
#define ZSI_STORE_64_1             LSTORE_1
#define ZSI_STORE_64_2             LSTORE_2
#define ZSI_STORE_64_3             LSTORE_3
#define ZSI_ARRAY_STORE_16         SASTORE
#define ZSI_ARRAY_STORE_32         IASTORE
#define ZSI_ARRAY_STORE_64         LASTORE
#define ZSI_RETURN_32              IRETURN
#define ZSI_RETURN_64              LRETURN
#define ZSI_DROP                   POP
#define ZSI_FLOAT_NEG              FNEG


#define ZSI_WIDE_LOAD_32      0xF9
#define ZSI_WIDE_LOAD_64      0xFA
#define ZSI_WIDE_STORE_32     0xFB
#define ZSI_WIDE_STORE_64     0xFC
#define ZSI_WIDE_RET          0xFD
#define ZSI_WIDE_INC_32       0xFE
```

```
#define OUTCODE         0
#define LENGTH          1
#define TABLE_LENGTH            0x100


char newCode[TABLE_LENGTH] [2]  = {
{NOP,                   1},  /*NOP              = 0x00,*/
{ZSI_ZERO_32,           1},  /*ACONST_NULL      = 0x01,*/
{ICONST_M1,             1},  /*ICONST_M1        = 0x02,*/
{ZSI_ZERO_32,           1},  /*ICONST_0         = 0x03,*/
{ICONST_1,              1},  /*ICONST_1         = 0x04,*/
{ICONST_2,              1},  /*ICONST_2         = 0x05,*/
{ICONST_3,              1},  /*ICONST_3         = 0x06,*/
{ICONST_4,              1},  /*ICONST_4         = 0x07,*/
{ICONST_5,              1},  /*ICONST_5         = 0x08,*/
{ZSI_ZERO_64,           1},  /*LCONST_0         = 0x09,*/
{LCONST_1,              1},  /*LCONST_1         = 0x0A,*/
{ZSI_ZERO_32,           1},  /*FCONST_0         = 0x0B,*/
{FCONST_1,              1},  /*FCONST_1         = 0x0C,*/
{FCONST_2,              1},  /*FCONST_2         = 0x0D,*/
{ZSI_ZERO_64,           1},  /*DCONST_0         = 0x0E,*/
{DCONST_1,              1},  /*DCONST_1         = 0x0F,*/


{BIPUSH,                2},  /*BIPUSH           = 0x10,*/
{SIPUSH,                3},  /*SIPUSH           = 0x11,*/
{LDC,                   2},  /*LDC              = 0x12,*/
{LDC_W,                 3},  /*LDC_W            = 0x13,*/
{LDC2_W,                3},  /*LDC2_W           = 0x14,*/
{ZSI_LOAD_32,           2},  /*ILOAD            = 0x15,*/
```

```
{ZSI_LOAD_64,            2},   /*LLOAD           = 0x16,*/
{ZSI_LOAD_32,            2},   /*FLOAD           = 0x17,*/


{ZSI_LOAD_64,            2},   /*DLOAD           = 0x18,*/
{ZSI_LOAD_32,            2},   /*ALOAD           = 0x19,*/
{ZSI_LOAD_32_0,          1},   /*ILOAD_0         = 0x1A,*/
{ZSI_LOAD_32_1,          1},   /*ILOAD_1         = 0x1B,*/
{ZSI_LOAD_32_2,          1},   /*ILOAD_2         = 0x1C,*/
{ZSI_LOAD_32_3,          1},   /*ILOAD_3         = 0x1D,*/
{ZSI_LOAD_64_0,          1},   /*LLOAD_0         = 0x1E,*/
{ZSI_LOAD_64_1,          1},   /*LLOAD_1         = 0x1F,*/
{ZSI_LOAD_64_2,          1},   /*LLOAD_2         = 0x20,*/
{ZSI_LOAD_64_3,          1},   /*LLOAD_3         = 0X21,*/
{ZSI_LOAD_32_0,          1},   /*FLOAD_0         = 0x22,*/
{ZSI_LOAD_32_1,          1},   /*FLOAD_1         = 0x23,*/
{ZSI_LOAD_32_2,          1},   /*FLOAD_2         = 0x24,*/
{ZSI_LOAD_32_3,          1},   /*FLOAD_3         = 0x25,*/
{ZSI_LOAD_64_0,          1},   /*DLOAD_0         = 0x26,*/
{ZSI_LOAD_64_1,          1},   /*DLOAD_1         = 0x27,*/


{ZSI_LOAD_64_2,          1},   /*DLOAD_2         = 0x28,*/
{ZSI_LOAD_64_3,          1},   /*DLOAD_3         = 0x29,*/
{ZSI_LOAD_32_0,          1},   /*ALOAD_0         = 0x2A,*/
{ZSI_LOAD_32_1,          1},   /*ALOAD_1         = 0x2B,*/
{ZSI_LOAD_32_2,          1},   /*ALOAD_2         = 0x2C,*/
{ZSI_LOAD_32_3,          1},   /*ALOAD_3         = 0x2D,*/
{ZSI_ARRAY_LOAD_32,      1},   /*IALOAD          = 0x2E,*/
{ZSI_ARRAY_LOAD_64,      1},   /*LALOAD          = 0x2F,*/
```

```
{ZSI_ARRAY_LOAD_32,        1},   /*FALOAD           = 0x30,*/
{ZSI_ARRAY_LOAD_64,        1},   /*DALOAD           = 0x31,*/
{ZSI_ARRAY_LOAD_32,        1},   /*AALOAD           = 0x32,*/
{BALOAD,                   1},   /*BALOAD           = 0x33,*/
{CALOAD,                   1},   /*CALOAD           = 0x34,*/
{SALOAD,                   1},   /*SALOAD           = 0x35,*/
{ZSI_STORE_32,             2},   /*ISTORE           = 0x36,*/
{ZSI STORE_64,             2},   /*LSTORE           = 0x37,*/

{ZSI_STORE_32,             2},   /*FSTORE           = 0x38,*/
{ZSI_STORE_64,             2},   /*DSTORE           = 0x39,*/
{ZSI_STORE_32,             2},   /*ASTORE           = 0x3A,*/
{ZSI_STORE_32_0,      1},   /*ISTORE_0         = 0x3B,*/
{ZSI_STORE_32_1,      1},   /*ISTORE_1         = 0x3C,*/
{ZSI_STORE_32_2,      1},   /*ISTORE_2         = 0x3D,*/
{ZSI_STORE_32_3,      1},   /*ISTORE_3         = 0x3E,*/
{ZSI_STORE_64_0,      1},   /*LSTORE_0         = 0x3F,*/


{ZSI_STORE_64_1,      1},   /*LSTORE_1         = 0x40,*/
{ZSI_STORE_64_2,      1},   /*LSTORE_2         = 0x41,*/
{ZSI_STORE_64_3,      1},   /*LSTORE_3         = 0x42,*/
{ZSI_STORE_32_0,      1},   /*FSTORE_0         = 0x43,*/
{ZSI_STORE_32_1,      1},   /*FSTORE_1         = 0x44,*/
{ZSI_STORE_32_2,      1},   /*FSTORE_2         = 0x45,*/
{ZSI_STORE_32_3,      1},   /*FSTORE_3         = 0x46,*/
{ZSI_STORE_64_0,      1},   /*DSTORE_0         = 0x47,*/
```

```
{ZSI_STORE_64_1,       1},  /*DSTORE_1        = 0x48,*/
{ZSI_STORE_64_2,       1},  /*DSTORE_2        = 0x49,*/
{ZSI_STORE_64_3,       1},  /*DSTORE_3        = 0x4A,*/
{ZSI_STORE_32_0,       1},  /*ASTORE_0        = 0x4B,*/
{ZSI_STORE_32_1,       1},  /*ASTORE_1        = 0x4C,*/
{ZSI_STORE_32_2,       1},  /*ASTORE_2        = 0x4D,*/
{ZSI_STORE_32_3,       1},  /*ASTORE_3        = 0x4E,*/
{ZSI_ARRAY_STORE_32, 1},    /*IASTORE         = 0x4F,*/

{ZSI_ARRAY_STORE_64, 1},    /*LASTORE         = 0x50,*/
{ZSI_ARRAY_STORE_32, 1},    /*FASTORE         = 0x51,*/
{ZSI_ARRAY_STORE_64, 1},    /*DASTORE         = 0x52,*/
{AASTORE,              1},  /*AASTORE         = 0x53,*/
{BASTORE,              1},  /*BASTORE         = 0x54,*/
{ZSI_ARRAY_STORE_16, 1},    /*CASTORE         = 0x55,*/
{ZSI ARRAY_STORE_16, 1},    /*SASTORE         = 0x56,*/
{ZSI_DROP,             1},  /*POP             = 0x57,*/


{POP2,                       1},  /*POP2            = 0x58,*/
{DUP,                        1},  /*DUP             = 0x59,*/
{DUP_X1,                     1},  /*DUP_X1          = 0x5A,*/
{DUP_X2,                     1},  /*DUP_X2          = 0x5B,*/
{DUP2,                       1},  /*DUP2            = 0x5C,*/
{DUP2_X1,                    1},  /*DUP2_X1         = 0x5D,*/
{DUP2_X2,                    1},  /*DUP2_X2         = 0x5E,*/
{SWAP,                       1},  /*SWAP            = 0x5F,*/
```

```
{IADD,                 1},  /*IADD,            = 0x60,*/
{LADD,                 1},  /*LADD             = 0x61,*/
{FADD,                 1},  /*FADD             = 0x62,*/
{DADD,                 1},  /*DADD             = 0x63,*/
{ISUB,                 1},  /*ISUB             = 0x64,*/
{LSUB,                 1},  /*LSUB             = 0x65,*/
{FSUB,                 1},  /*FSUB             = 0x66,*/
{DSUB,                 1},  /*DSUB             = 0x67,*/

{IMUL,                 1},  /*IMUL             = 0x68,*/
{LMUL,                 1},  /*LMUL             = 0x69,*/
{FMUL,                 1},  /*FMUL             = 0x6A,*/
{DMUL,                 1},  /*DMUL             = 0x6B,*/
{IDIV,                 1},  /*IDIV             = 0x6C,*/
{LDIV,                 1},  /*LDIV             = 0x6D,*/
{FDIV,                 1},  /*FDIV             = 0x6E,*/
{DDIV,                 1},  /*DDIV             = 0x6F,*/

{IREM,                 1},  /*IREM                   = 0x70,*/
{LREM,                 1},  /*LREM                   = 0x71,*/
{FREM,                 1},  /*FREM                   = 0x72,*/
{DREM,                 1},  /*DREM                   = 0x73,*/
{INEG,                 1},  /*INEG                   = 0x74,*/
{LNEG,                 1},  /*LNEG                   = 0x75,*/
{ZSI_FLOAT_NEG,        1},  /*FNEG                   = 0x76,*/
{ZSI_FLOAT_NEG,        1},  /*DNEG                   = 0x77,*/
```

```
{ISHL,          1},   /*ISHL        = 0x78,*/
{LSHL,          1},   /*LSHL        = 0x79,*/
{ISHR,          1},   /*ISHR        = 0x7A,*/
{LSHR,          1},   /*LSHR        = 0x7B,*/
{IUSHR,         1},   /*IUSHR       = 0x7C,*/
{LUSHR,         1},   /*LUSHR       = 0x7D,*/
{IAND           1},   /*IAND        = 0x7E,*/
{LAND,          1},   /*LAND        = 0x7F,*/

{IOR,           1},   /*IOR         = 0x80,*/
{LOR,           1},   /*LOR         = 0x81,*/
{IXOR,          1},   /*IXOR        = 0x82,*/
{LXOR,          1},   /*LXOR        = 0x83,*/
{IINC,          3},   /*IINC        = 0x84,*/
{I2L,           1},   /*I2L         = 0x85,*/
{I2F,           1},   /*I2F         = 0x86,*/
{I2D,           1},   /*I2D         = 0x87,*/

{ZSI_DROP,      1},   /*L2I         = 0x88,*/
{L2F,           1},   /*L2F         = 0x89,*/
{L2D,           1},   /*L2D         = 0x8A,*/
{F2I,           1},   /*F2I         = 0x8B,*/
{F2L,           1},   /*F2L         = 0x8C,*/
{F2D,           1},   /*F2D         = 0x8D,*/
{D2I,           1},   /*D2I         = 0x8E,*/
{D2L,           1},   /*D2L         = 0x8F,*/
```

```
{D2F,                       1},   /*D2F           = 0x90,*/
{I2B,                       1},   /*I2B           = 0x91,*/
{I2C,                       1},   /*I2C           = 0x92,*/
{I2S,                       1},   /*I2S           = 0x93,*/
{LCMP,                      1},   /*LCMP          = 0x94,*/
{FCMPL,                     1},   /*FCMPL         = 0x95,*/
{FCMPG,                     1},   /*FCMPG         = 0x96,*/
{DCMPL,                     1},   /*DCMPL         = 0x97,*/

{DCMPG,                     1},   /*DCMPG         = 0x98,*/
{ZSI_IF_ZERO_32,            3},   /*IFEQ          = 0x99,*/
{ZSI_IF_NOT_ZERO_32,        3},   /*IFNE          = 0x9A,*/
{IFLT,                      3},   /*IFLT          = 0x9B,*/
{IFGE,                      3},   /*IFGE          = 0x9C,*/
{IFGT,                      3},   /*IFGT          = 0x9D,*/
{IFLE,                      3},   /*IFLE          = 0x9E,*/
{ZSI_IF_EQUAL_32,           3},   /*IF_ICMPEQ     = 0x9F,*/


{ZSI_IF_NOT_EQUAL_32,       3},   /*IF_ICMPNE     = 0xA0,*/
{IF_ICMPLT,                 3},   /*IF_ICMPLT     = 0xA1,*/
{IF_ICMPGE,                 3},   /*IF_ICMPGE     = 0xA2,*/
{IF_ICMPGT,                 3},   /*IF_ICMPGT     = 0xA3,*/
{IF_ICMPLE,                 3},   /*IF_ICMPLE     = 0xA4,*/
{ZSI_IF_EQUAL_32,           3},   /*IF_ACMPEQ     = 0xA5,*/
{ZSI_IF_NOT_EQUAL_32,       3},   /*IF_ACMPNE     = 0xA6,*/
{GOTO,                      3},   /*GOTO          = 0xA7,*/
```

```
{JSR,                   3},  /*JSR               = 0xA8,*/
{RET,                   2},  /*RET               = 0xA9,*/
{TABLESWITCH,           0},  /*TABLESWITCH       = 0xAA,*/
{LOOKUPSWITCH,          0},  /*LOOKUPSWITCH      = 0xAB,*/
{ZSI_RETURN_32,         1},  /*IRETURN           = 0xAC,*/
{ZSI_RETURN_64,         1},  /*LRETURN           = 0xAD,*/
{ZSI_RETURN_32,         1},  /*FRETURN           = 0xAE,*/
{ZSI_RETURN_64,         1},  /*DRETURN           = 0xAF,*/

{ZSI_RETURN_32,         1},  /*ARETURN           = 0xB0,*/
{RETURN,                1},  /*RETURN            = 0xB1,*/
{GETSTATIC,             3},  /*GETSTATIC         = 0xB2,*/
{PUTSTATIC,             3},  /*PUTSTATIC         = 0xB3,*/
{GETFIELD,              3},  /*GETFIELD          = 0xB4,*/
{PUTFIELD,              3},  /*PUTFIELD          = 0xB5,*/
{INVOKEVIRTUAL,         3},  /*INVOKEVIRTUAL     = 0xB6,*/
{INVOKESPECIAL,         3},  /*INVOKESPECIAL     = 0xB7,*/


{INVOKESTATIC,          3},  /*INVOKESTATIC      = 0xB8,*/
{INVOKEINTERFACE,       5},  /*INVOKEINTERFACE   = 0xB9,*/
{UNUSED,                1},  /*UNUSED            = 0xBA,*/
{NEW,                   3},  /*NEW               = 0xBB,*/
{NEWARRAY,              2},  /*NEWARRAY          = 0xBC,*/
{ANEWARRAY,             3},  /*ANEWARRAY         = 0xBD,*/
{ARRAYLENGTH,           1},  /*ARRAYLENGTH       = 0xBE,*/
{ATHROW,                1},  /*ATHROW            = 0xBF,*/
```

```
{CHECKCAST,              3},  /*CHECKCAST          = 0xC0,*/
{INSTANCEOF,             3},  /*INSTANCEOF         = 0xC1,*/
{MONITORENTER,           1},  /*MONITORENTER       = 0xC2,*/
{MONITOREXIT,            1},  /*MONITOREXIT        = 0xC3,*/
{WIDE,                   0},  /*WIDE               = 0xC4,*/
{MULTIANEWARRAY,         4},  /*MULTIANEWARRAY     = 0xC5,*/
{ZSI_IF_ZERO_32,         3},  /*IFNULL             = 0xC6,*/
{ZSI_IF_NOT_ZERO_32,     3},  /*IFNONNULL          = 0xC7,*/

{GOTO_W,                 5},  /*GOTO_W             = 0xC8,*/
{JSR_W,                  5},  /*JSR_W              = 0xC9,*/
{BREAKPOINT,             1},  /*BREAKPOINT         = 0xCA,*/
{0xCB,                   1},
{0xCC,                   1},
{0xCD,                   1},
{0xCE,                   1},
{0xCF,                   1},


{0xD0,                   1},
{0xD1,                   1},
{0xD2,                   1},
{0xD3,                   1},
{0xD4,                   1},
{0xD5,                   1},
{0xD6,                   1},
{0xD7,                   1},
{0xD8,                  1},
{0xD9,                  1},
{0xDA,                  1},
{0xDB,                  1},
{0xDC,                  1},
{0xDD,                  1},
{0xDE,                  1},
{0xDF,                  1},

{0xE0,                  1},
{0xE1,                  1},
{0xE2,                  1},
{0xE3,                  1},
{0xE4,                  1},
{0xE5,                  1},
{0xE6,                  1},
{0xE7,                  1},


{0xE8,                  1},
{0xE9,                  1},
{0xEA,                  1},
{0xEB,                  1},
{0xEC,                  1},
{0xED,                  1},
{0xEE,                  1},
{0xEF,                  1},
{0xF0,                    1},
{0xF1,                    1},
{0xF2,                    1},
{0xF3,                    1},
{0xF4,                    1},
{0xF5,                    1},
{0xF6,                    1},
{0xF7,                    1},

{0xF8,                    1},
{0xF9,                    1},
{0xFA,                    1},
{0xFB,                    1},
{0xFC,                    1},
{0xFD,                    1},
{0xFE,                    1},  /*IMPDEP1          = 0xFE,*/
{0xFF,                    1}   /*IMPDEP2          = 0xFF*/
};
/*=====================================================
 * FUNCTION:   filterMethod
 * TYPE:    private operation on methods.
 * OVERVIEW:  Perform byte-code substitution on a given method.
 *
 * INTERFACE:
 *  parameters: this Method: method to be substituted.
 *  returns:  <nothing>
 *====================================================*/
```

```
void
filterMethod(METHOD thisMethod)
{
  unsigned short ip = 0; /*virtual ip */
  unsigned char *code = thisMethod->u.java.code;
  unsigned short codeLength = thisMethod->u.java.codeLength;

    while (ip < codeLength)
    {
   int opcode;

      opcode = code[ip];
      /* Filter all opcodes */
      code[ip] = newCode[opcode][OUTCODE];
      ip += newCode[opcode][LENGTH];

      /* Handle the following special opcodes */
      switch (opcode)
      {
        case TABLESWITCH:
          {
             long *lpc = (long*)(((long)(code + ip + 1) +3) &
~3);
              int cells;
             cells = getCell(&lpc[2]) - getCell(&lpc[1]) + 1 +
3;
             lpc += cells;
             ip = (unsigned char*)(lpc) - code;
             break;
        }
      case LOOKUPSWITCH:
        {
          long *lpc = (long *)(((long)(code + ip + 1) + 3) &
~3);
          int cells;

          cells = getCell(&lpc[1]) * 2 + 2;
          lpc += cells;
          ip = (unsigned char*)(lpc) - code;
          break;
        }
      case WIDE:
        code[ip] = NOP;
        ip++;
        switch (code[ip])
        {
          case ALOAD, FLOAD, ILOAD:
            {
              code[ip] = ZSI_WIDE_LOAD_32;
              ip += 3 ;
              break;
            }
          case DLOAD, LLOAD:
            {
              code[ip] = ZSI_WIDE_LOAD_64;
              ip += 3;
              break;
            }
```

```
 case ASTORE, FSTORE, ISTORE:
      {
        code[ip]= ZSI_WIDE_STORE_32;
        ip += 3;
        break;
      }
 case DSTORE, LSTORE:
      {
        code[ip]= ZSI_WIDE_STORE_64;
        ip += 3;
        break;
      }
 case IINC:
      }
        code[ip] = ZSI_WIDE_INC_32;
        ip += 5;
        break;
      }
 case RET:
      {
        code[ip] = ZSI_WIDE_RET;
        ip += 3;
        break;
      }
    }
    break;
default:
    break;
```

```
        }
    }
}
/*====================================================
 * FUNCTION:   filterClass
 * TYPE:  public operation on classes.
 * OVERVIEW:   Perform byte-code substitution of a given class.
 *        Iterate through all methods.
 *
 * INTERFACE:
 * parameters: thisClass: class to be filtered.
 * returns: <nothing>
 *===================================================*/
void
filterClass(INSTANCE_CLASS thisClass)
{
  int i;
  if (thisClass->methodTable){
   for (i = 0; i < thisClass->methodTable->length; i++) {
   METHOD thisMethod = &thisClass->methodTable->methods[i];
   /* Skip special synthesized methods. */

 if(thisMethod==RunCustomCodeMethod||thisMethod==unClinitMethod)
 {
      continue;
   }
   /* Skip abstract and native methods. */
   if(thisMethod->accessFlags &(ACC_NATIVE | ACC_ABSTRACT)) {
     continue;
```

```
      }
      filterMethod(thisMethod);
      }
     }
    }
   }
```

Where the implementations described have assumed a modified instruction processing machine having a reduced instruction set, and perhaps additional custom instructions, advantageously, a Java method transformed by the substitution method will still run correctly on a non-modified instruction processing machine.

Numerous modifications and variations of the present invention are possible in light of the above teachings. It is therefore to be understood that within the scope of the appended claims, the invention may be practiced otherwise than as specifically described herein.

WE CLAIM:

1. A processor implemented method of transforming a sequence of instructions of a platform independent instruction set, the method comprising:

defining a group of functionally equivalent instructions in the instruction set and defining a substitution instruction;

substituting the substitution instruction for each occurrence in the sequence of instructions of one of the predetermined group of functionally equivalent instructions.

2. A method according to claim 1 wherein the instruction set is the instruction set of an object oriented language.

3. A method according to claim 2 further comprising:

performing class loading;

wherein substituting the substitution instruction is done while performing class loading.

4. A processor implemented method of loading a Java-like method comprising:

defining a first group of functionally equivalent byte codes and defining a first substitution byte code;

substituting the first substitution byte code for each occurrence in the Java-like method of one of the first group of functionally equivalent byte codes.

5. A method according to claim 4 further comprising:

defining a plurality of groups of functionally equivalent byte codes one of which is said first group, and defining for each group of functionally equivalent byte codes a respective substitution byte code, one of which is said first substitution byte code;

for each occurrence in the Java-like method of a byte code in a respective one of the groups of functionally equivalent byte codes, substituting the substitution byte code defined for the respective one of the group of functionally equivalent byte codes.

6. A method according to claim 5 wherein for each group of functionally equivalent byte codes, the respective substitution byte code is one of the group of functionally equivalent byte codes.

7. A method according to claim 5 adapted to, for each occurrence in the Java-like method of a byte code in a respective one of the groups of functionally equivalent byte codes, substitute the substitution byte code defined for the respective one of the group of functionally equivalent byte codes by:

processing each byte code in the Java-like method in sequence to determine if the byte code is a byte code in one of the groups of functionally equivalent byte codes;

upon determining a byte code in the Java-like method is in one of the groups of functionally equivalent byte codes, substituting the substitution byte code;

after each processing of a byte code in the Java-like method, skipping over any subsequent bytes in the method which are arguments of the byte code, and continuing with processing a subsequent byte code following the bytes which are arguments of the byte code.

8. A method according to claim 4 wherein the first group of functionally equivalent byte codes comprises at least two byte codes selected from any one or groups group_1 through group_35 defined as follows:

```
group_1 = aload, fload, iload
group_2 = aload_0, fload_0, iload_0
group_3 = aload_1, fload_1, iload_1
group_4 = aload_2, fload_2, iload_2
group_5 = aload_3, fload_3, iload_3
group_6 = astore, fstore, istore
group_7 = astore_0, fstore_0, istore_0
group_8 = astore_1, fstore_1, istore_1
group_9 = astore_2, fstore_2, istore_2
group_10 = astore_3, fstore_3, istore_3
group_11 = dload, lload
group_12 = dload_0, lload_0
group_13 = dload_1, lload_1
group_14 = dload_2, lload_2
group_15 = dload_3, lload_3
group_16 = dstore, lstore
group_17 = dstore_0, lstore_0
group_18 = dstore_1, lstore_1
group_19 = dstore_2, lstore_2
group_20 = dstore_3, lstore_3
group_21 = if_acmpeq, if_icmpeq
group_22 = if_acmpne, if_icmpne
group_23 = ifeq, ifnull
group_24 = ifne, ifnonnull
group_25 = aconst_null, fconst_0, iconst_0
group_26 = dconst_0, lconst_0
group_27 = aaload, faload, iaload
group_28 = daload, laload
```

group_29 = fastore, iastore

group_30 = dastore, lastore

group_31 = castore, sastore

group_32 = areturn, freturn, ireturn

group_33 = dreturn, lreturn

group_34 = pop, l2i

group_35 = dneg, fneg

9. A method according to claim 4 further comprising:

defining for at least one group of functionally equivalent multi-byte instructions a respective substitution byte code;

for each occurrence in the Java-like method of a multi-byte instruction in a respective one of the at least one group of functionally equivalent multi-byte instructions, substituting the respective substitution byte code defined for the respective one of the at least one group of functionally equivalent multi-byte instructions.

10. A method according to claim 9 wherein the at least one group of functionally equivalent multi-byte instructions comprises at least two multi-byte instructions selected from any one of groups group_A through group_D defined as follows:

group_A = wide + aload or iload or fload

group_B = wide + lload or dload

group_C = wide + astore or istore or fstore

group_D = wide + lstore or dstore.

11. A method according to claim 4 further comprising:

defining for at least one multi-byte instruction a respective substitution byte code;

for each occurrence in the Java-like method of one of the at least one multi-byte instructions, substituting the respective substitution byte code.

12. A method according to claim 11 wherein the at least one multi-byte instruction is selected from a group consisting of:

wide + aload or iload or fload; wide + lload or dload; wide + astore or istore or fstore; wide + lstore or dstore; ret_w = wide + ret; wide + iinc.

13. A method of transforming a sequence of instructions in a platform independent instruction set including single instruction-width instructions and multi-instruction-width instructions, the method comprising:

defining for at least one multi-instruction-width instruction a respective substitution single-instruction-width instruction;

for each occurrence in the sequence of one of the at least one multi-instruction-width instructions, substituting the respective substitution single-instruction-width instruction.

14. A method according to claim 13 wherein the sequence of instructions is a sequence of Java-like byte codes.

15. A method according to claim 14 wherein the at least one multi-byte instruction is selected from a group consisting of:

wide + aload or iload or fload; wide + lload or dload; wide + astore or istore or fstore; wide + lstore or dstore; ret_w = wide + ret; wide + iinc.

16. A class loader adapted to load a Java-like method, the class loader comprising:

a semantic redundancy reduction substitution function adapted to define a plurality of groups of functionally equivalent byte codes and to define a respective substitution byte code for each group of functionally equivalent byte codes, and to substitute for each occurrence in the Java-like method of a byte code in one of the groups of functionally equivalent byte codes the substitution byte code defined for the group of functionally equivalent.

17. A processing platform comprising a class loader according to claim 16, and further comprising:

a Java-like machine containing a respective instruction implementation for each standard Java-like byte code not in said group of functionally equivalent byte codes, and containing an instruction implementation for the substitution byte code.

18. A processing platform according to claim 17 further comprising:

an instruction implementation for at least one byte code which is non-standard and which has not been used as a substitution byte code.

19. A processing platform according to claim 17 further comprising:

a class verifier adapted to perform verification functions on the Java-like method before substitution.

20. A processing platform according to claim 17 wherein the instruction implementations are all implemented in hardware.

21. A microprocessor comprising the processing platform according to claim 20.

22. A computer readable medium having instructions stored thereon for enabling a processor to implement a method according to claim 1.

23. A computer readable medium having instructions stored thereon for enabling a processor to implement a method according to claim 4.

24. An apparatus adapted to transform a sequence of instructions of a platform independent instruction set, the apparatus comprising:

a memory containing a definition of a group of functionally equivalent instructions in the instruction set and definition of a substitution instruction;

a processor adapted to substitute the substitution instruction for each occurrence in the sequence of instructions of one of the predetermined group of functionally equivalent instructions.

25. An apparatus according to claim 24 further comprising:

a class loading function executed by the processor, the class loading function having an input for receiving the sequence of instructions and having an output comprising a modified sequence of instructions in which the substitution instruction has been substituted.

26. An apparatus according to claim 25 adapted to transform a sequence of Java-like byte codes as said sequence of instructions, wherein each instruction is a Java-like byte code.

27. An apparatus according to claim 26 wherein:

the memory defines a plurality of groups of functionally equivalent byte codes one of which is said first group, and defines for each group of functionally equivalent byte codes a respective substitution byte code, one of which is said first substitution byte code;

the processor is adapted to, for each occurrence in the Java-like method of a byte code in a respective one of the groups of functionally equivalent byte codes, substitute the substitution byte code defined for the respective one of the group of functionally equivalent byte codes.

28. An apparatus according to claim 27 wherein the memory comprises:

a mapping from each possible input byte code to a corresponding transformed byte code, the corresponding transformed byte code being the same as the input byte code in the event no substitution is to be performed;

wherein the processor is adapted to process each byte code in the method in sequence by looking up the transformed byte code in the mapping, and replacing the byte code with the transformed byte code.

29. An apparatus according to claim 28 wherein the memory further comprises:

for each possible input byte code, an associated data entry indicating how many bytes should be skipped in order to arrive at a subsequent byte code;

wherein the processor is adapted to, after each processing of a byte code, skip to a subsequent byte code using the data entry in the memory associated with the input byte code

30. An apparatus according to claim 29 wherein the first group of functionally equivalent byte codes comprises at least two byte codes selected from any one or groups group_1 through group_35 defined as follows:

```
group_1 = aload, fload, iload
group_2 = aload_0, fload_0, iload_0
group_3 = aload_1, fload_1, iload_1
group_4 = aload_2, fload_2, iload_2
group_5 = aload_3, fload_3, iload_3
group_6 = astore, fstore, istore
group_7 = astore_0, fstore_0, istore_0
group_8 = astore_1, fstore_1, istore_1
group_9 = astore_2, fstore_2, istore_2
group_10 = astore_3, fstore_3, istore_3
group_11 = dload, lload
group_12 = dload_0, lload_0
group_13 = dload_1, lload_1
group_14 = dload_2, lload_2
group_15 = dload_3, lload_3
group_16 = dstore, lstore
group_17 = dstore_0, lstore_0
group_18 = dstore_1, lstore_1
group_19 = dstore_2, lstore_2
group_20 = dstore_3, lstore_3
group_21 = if_acmpeq, if_icmpeq
group_22 = if_acmpne, if_icmpne
group_23 = ifeq, ifnull
group_24 = ifne, ifnonnull
group_25 = aconst_null, fconst_0, iconst_0
group_26 = dconst_0, lconst_0
group_27 = aaload, faload, iaload
group_28 = daload, laload
```

group_29 = fastore, iastore

group_30 = dastore, lastore

group_31 = castore, sastore

group_32 = areturn, freturn, ireturn

group_33 = dreturn, lreturn

group_34 = pop, l2i

group_35 = dneg, fneg.

31. An apparatus according to claim 24 wherein the memory further comprises:

a definition for at least one group of functionally equivalent multi-instruction-width instructions of a respective substitution single-instruction-width instruction;

wherein the processor is further adapted to, for each occurrence in the sequence of instructions of a multi-instruction-width instruction in a respective one of the at least one group of functionally equivalent multi-instruction-width instructions, substitute the respective substitution single-instruction-width instruction defined for the respective one of the at least one group of functionally equivalent multi-instruction-width instructions.

32. An apparatus according to claim 31 adapted to transform a sequence of Java-like byte codes as said sequence of instructions, wherein each instruction is a Java byte code, wherein the at least one group of functionally equivalent multi-instruction-width instructions comprises at least two multi-byte instructions selected from any one or groups group_A through group_D defined as follows:

group_A = wide + aload or iload or fload

group_B = wide + lload or dload

group_C = wide + astore or istore or fstore

group_D = wide + lstore or dstore.

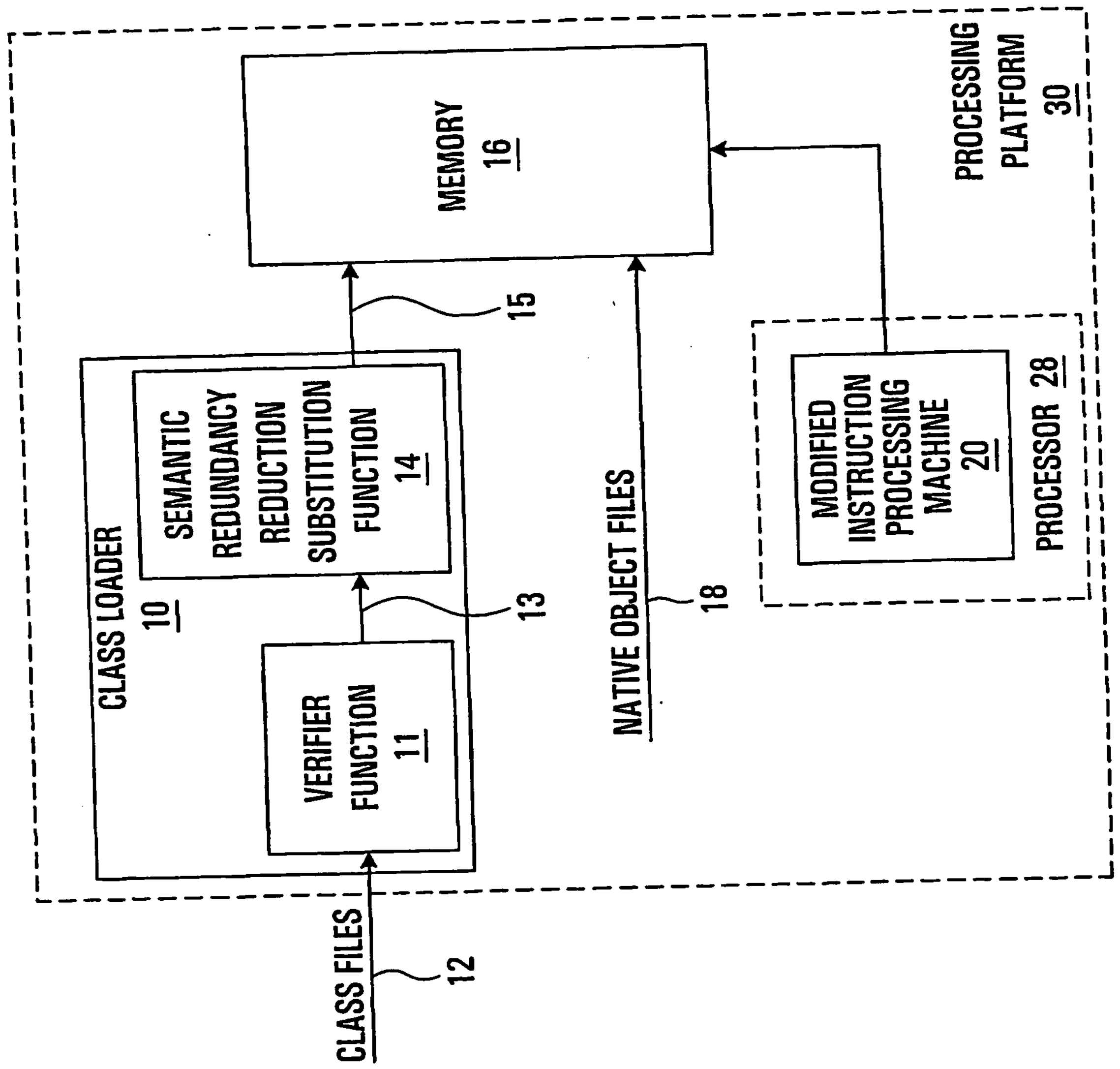

CLASS FILES
12
CLASS LOADER
10
VERIFIER FUNCTION
11
13
SEMANTIC REDUNDANCY REDUCTION SUBSTITUTION FUNCTION
14
15
MEMORY
16
NATIVE OBJECT FILES
18
MODIFIED INSTRUCTION PROCESSING MACHINE
20
PROCESSOR 28
PROCESSING PLATFORM
30

FIG. 1